(5)

PATENT ABSTRACTS OF JAPAN

(11)Publication number : 2000-029726
(43)Date of publication of application : 28.01.2000

(51)Int.Cl. G06F 9/46

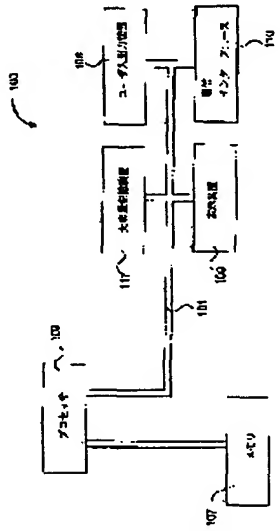
(21)Application number : 11-126259
(22)Date of filing : 06.05.1999

(71)Applicant : SUN MICROSYST INC
(72)Inventor : SHAYLOR NICHOLAS

(30)Priority
Priority number : 98 73637 Priority date : 06.05.1998 Priority country : US

(54) HIGH-SPEED SYNCHRONIZING METHOD FOR PROGRAMMING IN JAVA PROGRAMMING LANGUAGE

(57)Abstract:
PROBLEM TO BE SOLVED: To obtain a method for synchronized thread execution on a multi- thread processor by including a step, etc., for giving an indication so that a common object is to be locked by pushing the object identifier(OID) of the common object onto a lock stack, when a selected thread is entered.
SOLUTION: A computer system 100 is equipped with a system bus 101 and a processor 102. In this system, respective threads which are synchronized refer to an object which is shared by the threads and identified with OID. Then one of the threads is selected for execution. When the selected thread is entered, the OID of the common object is pushed on the lock stack to give an indication so that the common object is locked. Furthermore, the OID is pushed out of the lock stack for removing the indication.



LEGAL STATUS
[Date of request for examination]
[Date of sending the examiner's decision of rejection]
[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]
[Date of final disposal for application]
[Patent number]
[Date of registration]
[Number of appeal against examiner's decision of rejection]
[Date of requesting appeal against examiner's decision of rejection]
[Date of extinction of right]

(19)日本国特許庁（ＪＰ）　(12)公開特許公報（Ａ）　(11)特許出願公開番号
特開2000－29726
（P2000－29726A）
(43)公開日　平成12年１月28日(2000.1.28)

(51)Int.Cl.'	識別記号	ＦＩ		テーマコード'(参考)
Ｇ０６Ｆ　9/46	３４０	Ｇ０６Ｆ　9/46	３４０Ｂ	
			３４０Ｈ	

審査請求　未請求　請求項の数24　ＯＬ　外国語出願　（全 41 頁）

(21)出願番号　特願平11－126259

(22)出願日　平成11年５月６日(1999.5.6)

(31)優先権主張番号　０７３６３７

(32)優先日　平成10年５月６日(1998.5.6)

(33)優先権主張国　米国（ＵＳ）

特許法第64条第２項ただし書の規定により×印の部分は不掲載とした。

(71)出願人　597004720
サン・マイクロシステムズ・インコーポレーテッド
Ｓｕｎ　Ｍｉｃｒｏｓｙｓｔｅｍｓ，Ｉｎｃ．
アメリカ合衆国カリフォルニア州94303，パロ・アルト，サン・アントニオ・ロード　901，エムエス・ピーエイエル01－521

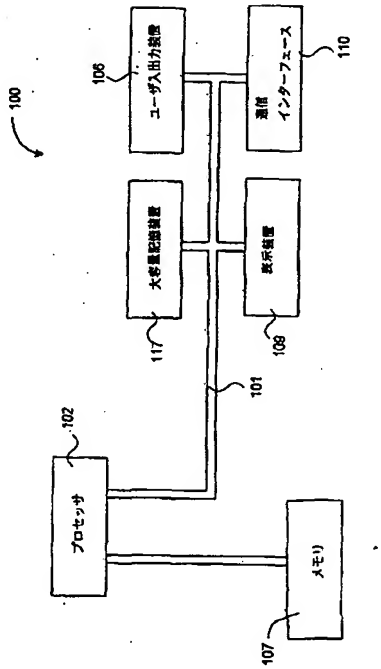
(72)発明者　ニコラス・シャイラー
アメリカ合衆国カリフォルニア州94560，ニューアーク，ボトレロ・ドライブ　6167

(74)代理人　100089705
弁理士　社本　一夫　（外５名）

最終頁に続く

(54)【発明の名称】　ＪＡＶＡプログラミング言語で書かれたプログラミングのための高速同期方法

(57)【要約】
【課題】　マルチスレッド・プロセッサにおいて同期のとれたスレッド実行のための方法を提供する。
【解決手段】　それぞれの同期のとれたスレッドは、複数の同期のとれたスレッドの間で共有されるオブジェクト識別子（ＯＩＤ）によって識別される少なくとも１つのオブジェクトを参照する。同期のとれているスレッドの中の１つが、実行のために選択される。選択されたスレッドに入る際に、エントリ・シーケンスは、共有されているオブジェクトが、そのＯＩＤをロック・スタックの上へのプッシュによってロックされるように指示する。選択されたスレッドによって定義される動作が実行され、ＯＩＤをロック・スタックからのポップによって、除去される。



【特許請求の範囲】

【請求項１】　複数のスレッドがその上で実行されているプロセッサにおける実行方法であって、前記スレッドは、少なくとも１つの共有オブジェクトを参照する同期のとれた動作を含み、前記共有オブジェクトは、オブジェクト識別子（ＯＩＤ）によって識別される、方法において、実行のための同期のとれた動作を含む前記複数のスレッドの中の第１のスレッドを選択するステップと、前記選択されたスレッドに入るときに、前記少なくとも1つの共有オブジェクトのＯＩＤをロック・スタックの上にプッシュすることによって、前記少なくとも１つの共有オブジェクトがロックされるように指示するステップと、前記ＯＩＤを前記ロック・スタックからプッシュすることによって、前記指示を除去するステップと、を含むことを特徴とする方法。

【請求項２】　請求項１記載の方法において、前記選択されたスレッド・ブロックによって定義された動作を実行するステップの間に、前記選択されたスレッド・ブロックの後に前記少なくとも１つの共有オブジェクトをロックするステップを更に含むことを特徴とする方法。

【請求項３】　請求項２記載の方法において、前記共有オブジェクトに対応するモニタ・オブジェクトの例を作成するステップを更に含むことを特徴とする方法。

【請求項４】　請求項３記載の方法において、前記モニタ・オブジェクトは、キュー・ヘッダと、カウンタと、前記選択されたスレッドを識別する所有者フィールドとを含むことを特徴とする方法。

【請求項５】　請求項３記載の方法において、前記ロックするステップは、アプリケーション・メモリ空間において前記モニタ・オブジェクトを作成するステップを含むことを特徴とする方法。

【請求項６】　請求項４記載の方法において、前記ロックするステップは、カーネル・メモリ空間において前記モニタ・オブジェクトを作成するステップを含むことを特徴とする方法。

【請求項７】　請求項２記載の方法において、前記ロックするステップは、更に、前記選択されたスレッドがいつブロックするかを判断するステップと、前記判断ステップに応答して、前記ロック動作を実行するステップと、を含むことを特徴とする方法。

【請求項８】　請求項２記載の方法において、前記ロックするステップは、更に、前記選択されたスレッドが前記ブロックの後でいつ再開するかを判断するステップと、同期のとれた動作を含む前記スレッドの別の１つが前記選択されたスレッドがブロックした後でブロックした場合にだけ前記ロック動作を実行するステップと、を含むことを特徴とする方法。

【請求項９】　請求項１記載の方法において、実行のために第２のスレッドを選択するステップであって、前記第２のスレッドは、前記第１のスレッドの同期のとれた動作と同期した動作を含む、ステップと、前記第２のスレッドに入るときに、前記共有オブジェクトがロックされているかどうかを判断するステップと、を更に含むことを特徴とする方法。

【請求項１０】　請求項９記載の方法において、前記共有オブジェクトがロックされていると判断されるときに、前記共有オブジェクトと関連付けされたモニタ・オブジェクトを作成して前記共有オブジェクトがいつロック解除されるかを判断するステップを更に含むことを特徴とする方法。

【請求項１１】　請求項９記載の方法において、前記共有オブジェクトがロック解除されていると判断されるときに、そのＯＩＤを前記ロック・スタックの上にプッシュすることによって、前記共有オブジェクトがロックされるように指示するステップと、前記第２のスレッドによって定義される動作を実行するステップと、前記ＯＩＤを前記ロック・スタックからプッシュすることによって、前記指示を除去するステップと、を更に含むことを特徴とする方法。

【請求項１２】　複数の同期のとれた実行スレッドを含むアプリケーションを実行するコンピュータ・システムであって、それぞれの同期のとれたスレッドは、複数の同期のとれたスレッドの間で共有されるオブジェクト識別子（ＯＩＤ）によって識別される少なくとも１つのオブジェクトを参照する、コンピュータ・システムにおいて、プロセッサと、前記プロセッサに結合されたメモリと、前記メモリの共有アドレス空間において複数の実行スレッドをサポートするマルチスレッド動作システムと、１つのＯＩＤを保持するサイズの複数のエントリを含む、前記メモリにおけるロック・スタックと、前記プロセッサにおいて動作し、選択された１つの同期のとれたスレッドを受け取るように結合され、前記選択されたスレッドを前記プロセッサ上で動作させる命令インタープリタであって、前記選択されたスレッドに入るときに、そのＯＩＤをロック・スタックの上にプッシュすることによって、前記共有オブジェクトがロックされるように指示し、前記選択されたスレッドから出るときに、前記ＯＩＤを前記ロック・スタックからプッシュすることによって、前記指示を除去する、命令インタープリタと、を備えていることを特徴とするコンピュータ・システム。

【請求項１３】　請求項１２記載のコンピュータ・システムにおいて、更に、前記プロセッサ内で動作し、前記選択されたスレッドが実行の間にいつブロックするかを告知するスレッド・ブロック・インジケータと、前記プロセッサ内で動作し、前記スレッド・ブロック・インジケータに応答して、前記ロック・スタックにおけるＯＩＤによって識別される前記共有オブジェクトをロックするオブジェクト・ロック装置と、を備えていることを特徴とするコンピュータ・システム。

【請求項１４】　請求項１２記載のコンピュータ・システムにおいて、前記オブジェクト・ロック装置は、前記共有オブジェクトに対応するモニタ・オブジェクトの例を備えており、前記モニタ・オブジェクトは、キュー・ヘッダと、カウンタと、前記選択されたスレッドを識別する所有者フィールドとを含むことを特徴とするコンピュータ・システム。

【請求項１５】　請求項１４記載のコンピュータ・システムにおいて、前記モニタ・オブジェクトは、前記メモリのアプリケーション・メモリ空間において例が与えられることを特徴とするコンピュータ・システム。

【請求項１６】　請求項１４記載のコンピュータ・システムにおいて、前記モニタ・オブジェクトは、前記メモリのカーネル・メモリ空間において例が与えられることを特徴とするコンピュータ・システム。

【請求項１７】　請求項１４記載のコンピュータ・システムにおいて、更に、前記プロセッサ内で動作し、前記選択されたスレッドが実行の間にブロック条件からいつリリースするかを告知するスレッド・リリース・インジケータと、前記プロセッサ内で動作し、前記スレッド・ブロック・インジケータに応答して、前記同期のとれたスレッドの別の１つが前記選択されたスレッドがブロックした後でブロックした場合にだけ、前記ロック・スタックにおける前記ＯＩＤによって識別される前記共有オブジェクトをロックするオブジェクト・ロック装置と、を備えていることを特徴とするコンピュータ・システム。

【請求項１８】　コンピュータ・プログラム製品であって、マルチスレッド・プロセッサにおいて同期のとれたスレッド実行のためにその中で実現されているコンピュータ読み取り可能なコードを有するコンピュータ使用可能な媒体であって、それぞれの同期のとれたスレッドは、複数の同期のとれたスレッドの間で共有されているオブジェクト識別子（ＯＩＤ）によって識別される少なくとも１つのオブジェクトを参照する、コンピュータ使用可能な媒体を備えているコンピュータ・プログラム製品において、コンピュータ・システムにおいて動作し、コンピュータに、前記同期のとれたスレッドの中の１つを実行のために選択させるように構成されたコンピュータ・プログラム・デバイスと、前記コンピュータ・システムにおいて動作し、コンピュータに、前記選択されたスレッドに入るときに、そのＯＩＤをロック・スタックの上にプッシュすることによって、前記共有オブジェクトがロックされるように指示させるように構成されたコンピュータ・プログラム・デバイスと、前記コンピュータ・システムにおいて動作し、コンピュータに、前記選択されたスレッドによって定義された動作を実行させるように構成されたコンピュータ・プログラム・デバイスと、前記コンピュータ・システムにおいて動作し、コンピュータに、前記ＯＩＤを前記ロック・スタックからプッシュすることによって、前記指示を除去させるように構成されたコンピュータ・プログラム・デバイスと、を備えていることを特徴とするコンピュータ・プログラム製品。

【請求項１９】　請求項１８記載のコンピュータ・プログラム製品において、更に、コンピュータに、前記選択されたスレッド・ブロックの後に前記共有オブジェクトをロックさせるように構成されたコンピュータ・プログラム・デバイスを備えていることを特徴とするコンピュータ・プログラム製品。

【請求項２０】　請求項１８記載のコンピュータ・プログラム製品において、更に、コンピュータに、前記共有オブジェクトに対応するモニタ・オブジェクトの例を作成させるように構成されたコンピュータ・プログラム・デバイスを備えており、前記モニタ・オブジェクトは、キュー・ヘッダと、カウンタと、前記選択されたスレッドを識別する所有者フィールドとを含むことを特徴とするコンピュータ・プログラム製品。

【請求項２１】　請求項１９記載のコンピュータ・プログラム製品において、更に、コンピュータに、前記選択されたスレッドがいつブロックするかを判断させるように構成されたコンピュータ・プログラム・デバイスと、コンピュータに、前記判断ステップに応答して、前記ロック動作を実行させるように構成されたコンピュータ・プログラム・デバイスと、を備えていることを特徴とするコンピュータ・プログラム製品。

【請求項２２】　請求項１９記載のコンピュータ・プログラム製品において、更に、コンピュータに、前記選択されたスレッドが前記ブロックの後でいつ再開するかを判断させるように構成されたコンピュータ・プログラム・デバイスと、コンピュータに、前記同期のとれたスレッドの別の１つが前記選択されたスレッドがブロックした後でブロックした場合にだけ前記ロック動作を実行させるように構成されたコンピュータ・プログラム・デバイスと、を備えていることを特徴とするコンピュータ・プログラム製品。

【請求項２３】　請求項１８記載のコンピュータ・プログラム製品において、更に、コンピュータに、共有オブジェクトと関連付けされたモニタ・オブジェクトを作成させるように構成されたコンピュータ・プログラム・デバイスであって、前記共有オブジェクトは、オブジェクト識別子（ＯＩＤ）によって識別され、前記モニタ・オブジェクトは、前記共有オブジェクトがロックされているときに、前記共有オブジェクトがいつロック解除されるかを判断する状態情報と方法とを含む、コンピュータ・プログラム・デバイスと、コンピュータに、前記第２のスレッドに入るときに、前記共有オブジェクトがロックされているかどうかを判断させるように構成されたコンピュータ・プログラム・デバイスと、前記共有オブジェクトがロックされている場合に、コンピュータに、前

記共有オブジェクトと関連付けされたモニタ・オブジェクトを作成して前記共有オブジェクトがいつロック解除されるかを判断させるように構成されたコンピュータ・プログラム・デバイスと、前記共有オブジェクトがロック解除されている場合に、コンピュータに、そのＯＩＤを前記ロック・スタックの上にプッシュすることによって、前記共有オブジェクトがロックされるように指示させるように構成されたコンピュータ・プログラム・デバイスと、コンピュータに、前記第２のスレッドによって定義される動作を実行させるように構成されたコンピュータ・プログラム・デバイスと、コンピュータに、前記ＯＩＤを前記ロック・スタックからプッシュすることによって、前記指示を除去させるように構成されたコンピュータ・プログラム・デバイスと、を備えていることを特徴とするコンピュータ・プログラム製品。

【請求項２４】　搬送波の中に具体化されているコンピュータ・データ信号であって、コンピュータに、共有オブジェクトと関連付けされたモニタ・オブジェクトを作成させるように構成されたコードを含む第１のコード部分であって、前記共有オブジェクトは、オブジェクト識別子（ＯＩＤ）によって識別され、前記モニタ・オブジェクトは、前記共有オブジェクトがロックされているときに、前記共有オブジェクトがいつロック解除されるかを判断する状態情報と方法とを含む、第１のコード部分と、前記共有オブジェクトがロック解除されている場合に、コンピュータに、そのＯＩＤを前記ロック・スタックの上にプッシュすることによって、前記共有オブジェクトがロックされるように指示させるように構成されたコードを含む第２のコード部分と、コンピュータに、前記第２のスレッドによって定義される動作を実行させるように構成されたコードを含む第３のコード部分と、コンピュータに、前記ＯＩＤを前記ロック・スタックからプッシュすることによって、前記指示を除去させるように構成されたコードを含む第４のコード部分と、を備えていることを特徴とするコンピュータ・データ信号。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】本発明は、広くは、データ処理に関し、更に詳しくは、ＪＡＶＡ言語プログラムにおけるスレッドの同期に関する。

【０００２】

【従来の技術】ＪＡＶＡ™（サンマイクロシステムズ社の商標）プログラミング言語は、この出願の出願人であるサンマイクロシステムズ社によって開発されたオブジェクト指向のプログラミング言語である。ＪＡＶＡプログラミング言語は、ネットワークされたアプリケーションのための言語及びプログラミング環境として、成功している。ＪＡＶＡプログラミング言語は、その多くの特徴のために魅力的であるが、その特徴には、プログラム・スレッドの同時的な実行に対する標準化されたサポートが含まれる。ＪＡＶＡプログラミング言語の同時性の特徴（concurrency features）は、言語（シンタクス）のレベルにおいて、そして、「スレッド」ライブラリを通じて、提供される。言語レベルでは、オブジェクトの方法は、「同期がとれている」と宣言されることがありうる。同期がとれていると宣言されたクラス内の方法は、同時にはランせず、「モニタ」の制御の下でランして、変数が一貫した状態（consistent state）に留まることを保証する。

【０００３】同期がとれた方法に入る又はその方法から出るたびに、ＪＡＶＡ言語は、オペレーティング・システム（Ｏ／Ｓ）カーネルへのコールが、スレッド同期資源を割り当てることを要求する。カーネルへのコールは、使用の際にＯ／Ｓに依存する、数百でなければ数十の命令を要求しうる。これと比較して、同期がとれている方法自体は、僅かに数ラインのコードを必要とするだけである。例えば、辞書ハッシュ・テーブル方法は、１０よりも少ない命令を用いて実現できるが、それを同期のとれた方法として実現するには、典型的なオペレーティング・システムにおいて、１００を超える命令が必要となる。従って、スレッドの同期は、多くのプログラムの実行時間を、著しく増大させる。

【０００４】このオーバヘッドは、マルチ・スレッディングを多く用いスレッド同期に依存するプログラムにおいて、要求される。しかし、このオーバヘッドは、シングル・スレッドのプログラムにおいては不要である。同様に、マルチスレッドのプログラムにおいても、多数のスレッドが、実際に、同期オーバヘッドなしに正確に実行されることもありうる。従って、これらの資源が必要とされるときにだけ、Ｏ／Ｓのスレッド管理資源配分と関連するオーバヘッドを負担するだけであるスレッド同期機構が必要とされている。

【０００５】オペレーティング・システムは、通常は、プリエンプタブル（preemptable）及び非プリエンプタブルという２つの方法の中の一方で、マルチスレディングをイネーブルする。プリエンプタブル・スレッドのオペレーティング・システム（例えば、××××やウィンドウズ（登録商標）／ＮＴ）は、１つのスレッドをイネーブルして別の同時に実行されているスレッドに割り込むＯ／Ｓ技術及び装置を含む。従って、任意の与えられた時間において、１つの実行されているスレッドは、それ自体が継続して実行されるのか、それとも、別のスレッドによってブロックされるのかを予測できない。よって、アプリケーションは、それ自身でスレッド同期を管理することができない。というのは、そのアプリケーションは、いつスレッドがブロックされるのかに関する予測を欠いているからである。プリエンプタブルなスレッドは、また、マルチプロセシング・マシンにおいて、スレッドの実行を複数のプロセッサの間で効率的に配分する際にも有用である。

【０００６】
【発明が解決しようとする課題】非プリエンプタブルなマルチスレディングは、スレッド実行のあるモードをサポートするより単純な形式のマルチスレディングであり、いったんスレッドが実行され始めると、それは、別のスレッドにブロックされることはない。スレッドは、停止したりそれ自身をブロックし、他のスレッドに対する制御を生じたり、入出力（Ｉ／Ｏ）が完了するのを待機するためにブロックされることはありうる。シングル・スレッドとして実現することができオペレーティング・システムのプリエンプティブなマルチスレディングの特徴を必要としない多数のアプリケーションも存在する。非プリエンプティブなオペレーティング・システムも情報機器やより単純なオペレーティング・システムにおいては、存在しうる。それぞれのスレッドを先取りするのは不可能であることをＯ／Ｓが確認している場合には、Ｏ／Ｓにおけるスレッド同期資源の配分は、プログラムの実行時間を増大させるが、これには、ほとんど利点がない。
【０００７】マルチスレディングは、多くのプログラムに内在する（又は、その中に設計上組み込まれている）並列性（parallelism）を利用している。しかし、古くからある（legacy）プログラムは、多くの場合、ほとんど並列性を示さない。また、プログラムによっては、その振る舞いの性質上、並列性をそれほど示さないものもある。これらのプログラムは、その非並列的な構造に起因して有している利点を得ることなく、マルチスレディング・オペレーティング・システムに関連するオーバヘッドを負担しなければならないことによって、速度が低下する。従って、本質的にスレッドを有しない（un-threaded）が、マルチスレディング・オペレーティング・システム上で動いているプログラムにおいて、実行を高速化するスレッド同期機構に対する必要性が存在している。
【０００８】
【課題を解決するための手段】簡潔に述べると、本発明は、マルチスレッド・プロセッサにおいて同期のとれたスレッド実行のための方法に関する。それぞれの同期のとれたスレッドは、複数の同期のとれたスレッドの間で共有されるオブジェクト識別子（ＯＩＤ）によって識別される少なくとも１つのオブジェクトを参照する。同期のとれているスレッドの中の１つが、実行のために選択される。選択されたスレッドに入る際に、エントリ・シーケンスは、共有されているオブジェクトが、そのＯＩＤをロック・スタックの上にプッシュすることによって、ロックされるように指示する。選択されたスレッドによって定義される動作が実行され、ＯＩＤをロック・スタックからポップすることによって、除去される。
【０００９】本発明の以上の及びそれ以外の特徴、効用及び効果は、添付の図面において図解されている本発明の好適実施例についての以下で行うより詳細な説明から明らかになるはずである。本発明の更に別の実施例は、この技術分野の当業者にとっては、以下の詳細な説明から容易に明らかであろう。詳細な説明では、本発明の実施例が、本発明を実現するために考慮される最良の態様を図解することによって、説明されているだけである。理解すべきは、本発明は、本発明の精神と範囲とから逸脱することなく、様々な明らかな点において修正を行うことができるということである。従って、図面と詳細な説明とは、性質上、制限的なものではなく、例示的なものであると考えるべきである。
【００１０】
【発明の実施の態様】一般に、本発明は、ＪＡＶＡプログラミング言語で書かれており、プログラムの正しさを確認することが必要であると判断されるまで、オーバーヘッドの高いスレッド同期構造を始動させることを延期する進行中の同期のとれたスレッドを扱う方法に関する。換言すると、多くのアプリケーションにおいて、特にシングル・スレッドのプログラムにおいて、アプリケーション自身が、プログラムが決定論的（deterministic）であることを確認し、マルチスレッドの同期に関連する高いオーバーヘッドは、プログラムの実行を遅くするようにしか機能しない。従って、本発明によると、完全なデータ構造を含む複雑な同期機構の実現は、それらが実際には決して実現されないように、延期される。
【００１１】スレッド同期という問題は、ＪＡＶＡプログラミング言語に一意的なものではないが、ＪＡＶＡプログラミング言語のいくつかの問題が、この問題に影響する。プログラムが「同期がとれている」とラベル付けされた方法に入るたびに作成されたＯ／Ｓ指定されたロックやモニタ・オブジェクトを含んでいた。モニタ・オブジェクトは、かなり大きく、２つのキュー・ヘッダとカウンタと所有者フィールド（owner field）とを含むデータ構造を必要とする。これらの構造は、そのサイズのために、標準的なオブジェクト・ヘッダに含まれることができない。本発明は、これらのロック及びモニタ・オブジェクトの作成を必要となるまで回避することによって、この困難を解決する。
【００１２】本発明は、ＪＡＶＡプログラミング環境での実現例の言葉で説明される。この例には、ＪＡＶＡ仮想マシン（ＪＶＭ）、ジャストインタイム（ＪＩＴ）コンパイラ、ＪＡＶＡプログラミング言語コードを別のプログラミング言語に変換するコンパイル時間又はラン時間機構などがある。しかし、本発明は、ステートメント又はプログラム・オブジェクトがオペレーティング・システムのマルチスレディングの特徴にアクセスすることを可能にする任意のプログラミング言語において有用である。本発明は、プログラミング言語自体の変更を必要とするのではなく、プログラミングがスレッド同期をサポートするのに不必要なＯ／Ｓ動作を実行することを強

制されないようにプログラムを実現する方法を提供する。

【００１３】図１は、本発明による装置及びシステムを組み入れたコンピュータ・システムを、ブロック図形式で図解している。プロセッサ・アーキテクチャと計算システムとは、図１に示されているように、相互に作用する機能ユニットの集合として、便宜的に表されている。これらの機能ユニットは、メモリから命令及びデータをフェッチし、フェッチされた命令を処理し、メモリ・トランザクションを管理し、外部Ｉ／Ｏとのインターフェースを行い、情報を表示するという機能を実行する。

【００１４】図１には、プロセッサ１０２を組み入れており、プロセッサ１０２においてアプリケーション・プログラムとオペレーティング・システムとの両方を用いている典型的な汎用コンピュータ・システム１００が示されている。本発明によるコンピュータ・システム１００は、情報を通信するシステム・バス１０１と、プロセッサ１０２の中の入出力（Ｉ／Ｏ）装置を介してバス１０１に結合されたプロセッサ１０２とを備えている。プロセッサ１０２は、メモリ・バス１０３を用いて、プロセッサ１０２のための情報や命令を記憶するメモリ・システム１０７に結合される。メモリ・システム１０７は、例えば、１又は複数のレベルのキャッシュ・メモリと、メイン・メモリとを、メモリ・ユニット１０７において、備えている。メモリ・システム１０７内のキャッシュ及びメモリに加えて、多くの応用例では、いくらかのキャッシュ・メモリが、プロセッサ１０２と共にオンチップで含まれていることも理解すべきである。

【００１５】ユーザＩ／Ｏ装置１０６は、バス１０１に結合され、適切な構造を有する情報を、コンピュータ１００の他の部分との間で通信するように動作する。ユーザＩ／Ｏ装置は、キーボード、マウス、時期又はテープ・レコーダ、光ディスク、又は、別のコンピュータを含む他の利用可能なＩ／Ｏ装置を含みうる。大容量記憶装置１１７がバス１０１に結合され、１又は複数の磁気ハードディスク、磁気テープ、ＣＤ－ＲＯＭ、ランダム・アクセス・メモリの大きなバンクなどを用いて実現することができる。広範囲なランダム・アクセス及びリード・オンリ・メモリ技術が利用可能であり、本発明の目的に対しては、同等である。大容量記憶装置１１７は、その中に、コンピュータ・プログラムとデータとが記憶されている。大容量記憶装置１１７の一部又は全体は、メモリ・システム１０７の一部として組み込まれているようにも構成が可能である。

【００１６】本発明は、特に、Ｏ／Ｓスレッドがプリエンプタブルではないオペレーティング・システムにおいて、有用である。しかし、プリエンプタブルなスレッドを有するオペレーティング・システムでも、強制的に一度にただ１つのスレッドを実行させることによって、本発明と共に用いることが可能である。更に、本発明は、タスクごとにプリエンプタブルなスレッドと非プリエンプタブルなスレッドとの両方を扱うオペレーティング・システムでも用いることができる。

【００１７】多くのソフトウェア言語（ＪＡＶＡプログラミング言語を含む）におけるスレッド同期のための基本的な方法は、モニタ及びロック方式（monitor and lock paradigm）である。本発明はこのロッキング・プロセスを単純化しているが、本発明によって機能的に達成されるものは、スレッドが予測可能な態様で振る舞うように言語仕様を満足していなければならない。ロック動作は、まず最初に、特定されたオブジェクト（例えば、ここでの例におけるオブジェクトｘ）上でのロック取得を試みることから開始する。ロック動作によって、このロック動作が終了する（すなわち、ロックが得られる）までそれ以上のスレッドの実行を防止する。ロック動作が終了した後にだけ、同期のとれた方法のボディにおけるステートメントが実行される。与えられたスレッドのボディにおけるすべてのステートメントの実行が終了する、又は、そのスレッドがそれとは異なる事情で終了するときに、ロック解除動作が同じロックに対して実行される。

【００１８】従来技術によるロック機構に関連するＯ／Ｓオーバーヘッドは、もちろん、オブジェクトを全くロックしないことによって、又は、可能な限り少数のオブジェクトしかロックしないことによって、単純に回避することができる。しかし、任意の２つの方法が１又は複数のオブジェクトにアクセスする場合には、ロック機構を用いることが本質的である。本発明は、アプリケーションにおいて特定されたロック機構を効率的に実現して予測されるアプリケーションの振る舞いを提供する方法に向けられている。

【００１９】同期のとれたスレッドから成るグループの中のただ１つのスレッドだけが、図２の例において示されているように、一度に実行され得る場合には、一般的には、オブジェクトがロックされているときに関連するモニタ・データ構造を作成する必要は存在しない。同様に、複数のスレッドを実行することが可能であるが別のものの実行を先取りすることが不可能である場合には、モニタ・データ構造は一般に必要とされない。このような状況では、モニタ・データ構造は、実行されているスレッドがアプリケーション自体には未知の態様である時間の間ブロックされるときにだけ、必要となる。

【００２０】図2は、本発明によって生成されるスレッド２００を概略的に示している。それぞれのソフトウェア・アプリケーションは、特定のアプリケーションの振る舞いを実現することに向けられているタスクの集合から構成されている。そして、それぞれのタスクは、図２に示されているスレッド２００などの複数のスレッドによって実現されている。それぞれのスレッドは、それが実行しているプログラムのセマンティクスによって命じ

られる使用、割り当て、ロック及びロック解除動作のストリームを生じる。それぞれの単位となる動作は、図2においては、スレッド200内のボックスによって表されている。

【0021】本発明を理解する上で特に重要なのは、動作ボックスの中において、「L」によって示されているロック動作と、「U」によって示されているロック解除（アンロック）動作と、クロスによって示されているブロック動作と、である。参照を容易にするために、ブロック動作は、一連の動作ボックスによって示されており、ブロックの継続時間を示している。しかし、実際には、継続時間の全体に対してブロック動作を行うのは、1回の動作による。それ以外の動作は、空欄である動作ボックスによって一般的に示されているが、これらは、広範囲の動作を含みうる。スレッド200は、任意の数の単位となる動作を含むことができる。これは、本質的にシングルスレッドのタスクにおいて、10よりも少ない動作から数百、数千の動作までの幅がある。図2の楕円は、典型的なスレッド200には、任意の数の非ブロック及びブロック動作が含まれうることを示している。

【0022】マルチスレッド・アプリケーション又はタスクでは、タスクの振る舞いを実行するために、複数のスレッド200が実現されている。マルチスレッド・アプリケーションは、「シリアル」な態様で実行されるが、この意味は、スレッドがプリエンプタブルではない、ということである。既に述べたように、これは、オペレーティング・システムが命じることができる。

【0023】オペレーティング・システムがプリエンプタブルなスレッドを許容する場合には、本発明は、図3から図5に示されている同期のとれているスレッド0及びスレッド1が、同時に実行されないことを保証するように動作する。これは、例えば、それぞれの同期のとれた方法を「クリティカル・セクション」と称される構成で実行し、それによって、プログラム制御が同期のとれた方法を実行するよりも先にクリティカル・セクションに入るようにし、第1のスレッドが出口に入るまでそれ以外のスレッドがクリティカル・セクション（臨界部分）に入らないようにすることによって、なされる。

【0024】図3から図5は、スレッド0及びスレッド1とラベル付けされた同期のとれたスレッドの実行の間の様々な状況を図解しており、図2に関して上述したと同じ符号を用いている。図3は、スレッド0が多数の動作を実行し、その中の少なくとも1つが共有されたオブジェクト（図示せず）にアクセスしているという状況を示している。図3において、スレッド0は、スレッド1と同期している。というのは、例えば、スレッド1もまた、同じ共有されたオブジェクトにアクセスしているからである。スレッド0とスレッド1とは、生産者と消費者として関係することがあり得る（すなわち、一方のスレッドが共有されたオブジェクトを変更すると、他方のスレッドは、その変更されたオブジェクトを用いる）し、共有されたオブジェクトが一度にただ1つのアクセスだけをサポートし、それによって、スレッド0とスレッド1とが、衝突を避けるように同期することもありうる。

【0025】それぞれのスレッドにおけるロック及びロック解除動作は、そのスレッドのクリティカル・セクションに入る又はそこから出る際に、スレッドの実行の間の任意の時期に実行される（すなわち、共有されているオブジェクトにアクセスする動作を含むスレッド内の方法）。スレッドは、ロック解除動作を実行する前に、複数のロック動作を実行することがありうる（図2に示されているように）が、それぞれのロックの後には、共有されているオブジェクトが別のスレッドに対して利用可能となる前に、ロック解除がなされなければならない。プログラミング言語が、明白なオブジェクトのロック／ロック解除を許容することがある。また、JAVAプログラミング言語の場合には、この方法における「同期のとれた」ステートメントが、ロック／ロック解除動作を意味する。

【0026】ロック動作は、本発明に従って、オブジェクト識別子（OID）を専用のスタック（図7の701）の上にプッシュすることによって、実現される。ここでの特定の例では、OIDは、メモリにおける共有されているオブジェクト「x」のアドレスである。ただし、1又は複数のロックされたオブジェクトを一意的に識別する任意のOIDは、特定の実現例と同等である。従来技術の場合とは対照的に、ロック動作は、O／Sに資源を配分させたり、スレッドの同期を与えるデータ構造を生じたりすることはない。スレッドは、このようにして、複数のロックを取得し得る。ただし、単純にするために、ただ1つのロックだけを示してある。更に、スレッドは、同じスレッド上で複数のロックを取得し、それによって、それぞれの取得されたロックが、そのオブジェクトが別のスレッドに対して利用可能となる前にロック解除されなければならないようにすることができる。

【0027】実際のモニタ・オブジェクトが存在する場合にだけ、実行される任意の評価可能な（appreciable）コードが存在する必要がある。ロックが既に付与されている場合には、これによって、実行されているスレッドがブロックされる。同様の手順がロック解除時に実行され、スレッドが再開される。モニタ・オブジェクトの存在は規則ではなく例外であることを考慮すると、同期のとれた方法へのエントリ・シーケンスは、論理的には、次のようになる。

【0028】

【数1】

```
if(monitorExists(o)  ) {
     monitorLockProcessing(o)
} else {
     pushOnStack(o) ;
}
```

【００２９】そして、例示的な出口シーケンスは、次のようになる。

【００３０】

【数２】

```
if(monitorExists(o)  ) {
     monitorUnlockProcessing(o)
} else  {
     popStack();
}
```

【００３１】ただし、ここで、monitorLockProcessingと、monitorUnlockProcessingとは、通常のロック／ロック解除動作をコールする方法を表している。本発明による同期化技術は、スレッド０及びスレッド１の実行に対して順序付けの制約を課していないことに注意することが重要である。換言すると、スレッド１は、どの例においても、スレッド０よりも前に実行されることが可能であり、順序の制約を課すには、他の機構を用いなければならない。ＪＡＶＡプログラミング言語におけるこれらの機構の例としては、wait()、notify()、notifyAll()などの方法が。ＪＡＶＡプログラミング言語のスレッド・クラスにおいて利用可能である。ただし、それ以外のプログラム言語では、同等なものが利用可能である。

【００３２】図３の例では、スレッド１及びスレッド０は、ＣＰＵ拘束（CPU bound）ではない。これは、終了されなければならない動作の数と比較して、多くの利用可能なＣＰＵサイクルが存在していることを意味する。換言すると、スレッド０もスレッド１も、共有されている変数にアクセスし従って同期がとれていると宣言されているにもかかわらず、その実行の間は、相互に衝突することはない。ほとんどの命令を実行するのに必要なクロック・サイクルの数は、かなり予測可能である。ただし例外として、メモリ動作、入出力（Ｉ／Ｏ）を必要とする動作、特定のイベントが発生するのを待機する命令などは、この限りではない。図３に示されているように、スレッドの同期は、それほどの問題ではない。その理由は、スレッド１とスレッド０とは同時に実行されることはないからである。この場合には、Ｏ／Ｓスレッド同期資源は、不要である（ただし、従来技術では、これらは、配分されている）。本発明の著しい特徴は、図３に示されている状況の結果として、アプリケーションが同期スレッドを指定した場合であっても、Ｏ／Ｓ資源の支出（すなわち、Ｏ／Ｓモニタ・オブジェクトの作成）を生じさせないことである。

【００３３】図４は、関連の状況を示しているが、ここでは、実行の間にオブジェクトがロックされている間にスレッド０とスレッド１とがブロックする場合であっても、これらは、実際に衝突することはない。スレッド０におけるブロックは、動作６と動作９との間において発生し、例えば、Ｉ／Ｏ動作によって生じ得る。動作５においては、ロックは、共有されているオブジェクト上に配置され、従って、ブロック・フェーズの間は、共有されているオブジェクトはロックされたまま維持される。しかし、スレッド１は、動作１８におけるそれ自身のロック動作までは、共有されているオブジェクトへのアクセスを試みないので、衝突は存在しない。先の例におけるように、通常の処理では、Ｏ／Ｓ資源が消費され、実際には何も必要ではないにもかかわらず、時間５においてモニタ・オブジェクトが作成される。

【００３４】本発明の第１の実施例によると、共有されているオブジェクトのＯＩＤは、ロックがなされた（すなわち、時間６において）後でスレッド０が最初にブロックするときに、スタック７０１（図７に示されている）の上にプッシュされる。スレッドが時間１０において再開するときには、ＯＩＤが除去される。この第１の実施例の結果として、最小の資源の消費が生じる。その理由は、ＯＩＤをスタック７０１の上に単純にプッシュすることによって、Ｏ／Ｓへのコールなしに、ロックを実現できるからである。第２の実施例によると、システムは、１）スレッド０が再開し、２）再開したスレッドが最後にブロックしたスレッドにならないまで、ＯＩＤのスタック７０１上へのプッシュを遅延させる。従って、第２の実施例では、図４の例では、ロックは、決して実現されることはない。その理由は、スレッド０が、任意の他のスレッドがロックを作成する前に再開するからである。

【００３５】これと対照的に、図５は、その実行の間に動作８においてブロックするスレッド０を含む例を示している。図５では、スレッド０におけるブロック動作は、動作１８において再開する。動作１０において、スレッド１は、このスレッドが共有されているオブジェクトに対してロックを取得しようとしている動作１４まで、ブロック解除し、実行を再開する。この場合には、共有されているオブジェクトは、スレッド０によって既にロックされており、通常のロック／モニタ処理が、スレッド１にいつ継続が可能であるかを告知するのに必要である。本発明によると、スレッド１は、ボールド・ラインによるボックス１５－２２によって指示されている時間の間に作成されるモニタ・オブジェクトによって、ブロックされる。時間２３では、スレッド０が再びブロックするが、このときには、共有されているオブジェクトはロック解除され、スレッドの再開を可能にする。

【００３６】第１の実施例では、スレッド同期データ構

造の配分は、スレッド０がブロックする時間８での動作まで、遅延される。第２の実施例によると、ＯＩＤは、スレッド１が時間１０においてブロック解除するまで、スタック７０１の上には配置されない。従って、ロック／モニタ資源が実際に必要であるときにだけ、資源は配分される。第１の実施例では、資源配分の遅延は別として、スレッド０及びスレッド１は、従来の同期方法と実質的に同一の態様で実行される。第１の実施例は、パフォーマンスが改善されているが、これは、同期資源が、図３及び図４に示されているのと類似する状況においては、全く配分されていないからである。

【００３７】第２の実施例では、ロック／モニタ資源の配分は、スレッドの再開まで延期される。この場合には、スレッド１は、アプリケーション自身によってブロックされるのであるが、その理由は、オブジェクト上のロックを知っており、スタック７０１がアプリケーション又はユーザのメモリ空間に維持されているからである。スレッド１は、時間１０において再開すると、それがＯ／Ｓへのコールな痔でブロックした最後のスレッドであるかどうかを判断する。図５では、スレッド１はブロックした最後のスレッドではないので、モニタ・オブジェクトが、スレッド１をスレッド０と同期させるために必要となる。しかし、図４の例では、スレッド０が再開するとそれがブロックした最後のスレッドであるから、モニタ資源は要求されない。同期資源の配分を長く遅延させることが可能であればあるほど、衝突するスレッドの一方が、共有されているオブジェクトをロック解除し、同期が不要となる可能性が大きくなる。

【００３８】本発明の別の効果は、オペレーティング・システム資源とは無関係に、モニタ資源を実現することができるということである。図５との関係で説明したように、動作１０におけるスレッド１のブロック解除は、Ｏ／Ｓなしで扱うことができる。その理由は、共有されているオブジェクトがロックされているという情報が、アプリケーションのメモリ空間に保持されているからである。本発明は、通常のＯ／Ｓ同期資源を用いて実現することができるが、モニタ・オブジェクトをアプリケーションのメモリ空間内に実現し、モニタ・オブジェクトをＯ／Ｓ同期資源を用いてカーネル・メモリ空間において実現するのに要求される可能性のある数十又は数百の動作を回避することが望まれる。

【００３９】図６は、本発明を実現する基本的なソフトウェア又はハードウェアで実現された装置を示している。コンピュータ・システム１００（図１を参照）は、インタプリタ６０２を用いてアプリケーション・コード６０１を実行する。アプリケーション・コード６０１は、ＪＡＶＡプログラミング言語などの通常の言語における、シングル・スレッド又はマルチスレッドのコードである。インタプリタ６０２は、アプリケーション・コードを受け取り、実行エンジン６０４によって実行され得る実行可能な命令を発生する。この実行可能な命令は、Ｏ／Ｓ６０６へのコールを含むか、又は、メモリ１０７内のデータを直接に操作することができる。インタプリタ６０２は、ランタイムの前に（すなわち、コンパイル時間に）動作可能なコンパイラの形式で実現が可能であり、又は、ＪＡＶＡ仮想マシンと共に提供されるジャストインタイム・コンパイラなどのように、ランタイムにおいて動作可能である。インタプリタ６０２からの実行可能な命令は、ランタイムにおいて実行エンジン６０４によって後で実行されるように、記憶装置６０３（例えば、磁気媒体、光媒体など）にも記憶される。

【００４０】図７を参照すると、複数のオブジェクト７０２、７０３、７０５が示されている。オブジェクト７０３は、スタック７０１における対応するエントリによってロックされる。対照的に、オブジェクト７０５は、通常の態様で作成されたモニタ・オブジェクト７０４によってロックされる。オブジェクトがロックされたという事実と、ロックされた回数とは、オブジェクトのアドレス（図７において、オブジェクトＩＤ又はＯＩＤとして示されている）ををロック時間においてスタック７０１の上にプッシュすることによって、実行エンジン６０４によって記録される。エントリは、ロック解除時間において、スタック７０１からポップされ、ポップされたＯＩＤと関連するオブジェクトは、ロック解除されたオブジェクト７０２となる。このようにして、オブジェクトがロックされている間にスレッドがブロックしない場合には、スレッドの同期を保証するのに、これ以上の何も必要でない。しかし、スレッドがブロックする場合には、スタック７０１上のオブジェクトに、伝統的な方法で用いられるモニタ・データ構造７０４が配分され、スタック７０１が空にされる。

【００４１】既に述べたように、実際のロック／モニタ・データ構造が作成される希な場合には、それらは、オペレーティング・システム６０６とは無関係に、実現することができる。メモリ１０７は、アプリケーションすなわちユーザ・メモリ空間７０１とカーネルすなわちＯ／Ｓメモリ空間の中に配分される。

【００４２】以上で本発明をある程度まで特定して説明し図解したが、この開示は、単なる例示であり、冒頭の特許請求の範囲によって定義される本発明の精神と範囲とから逸脱することなく、部分的な組合せと配列とを行うことによって、多くの変更を行うことができることは、この技術分野の当業者には明らかである。

【図面の簡単な説明】

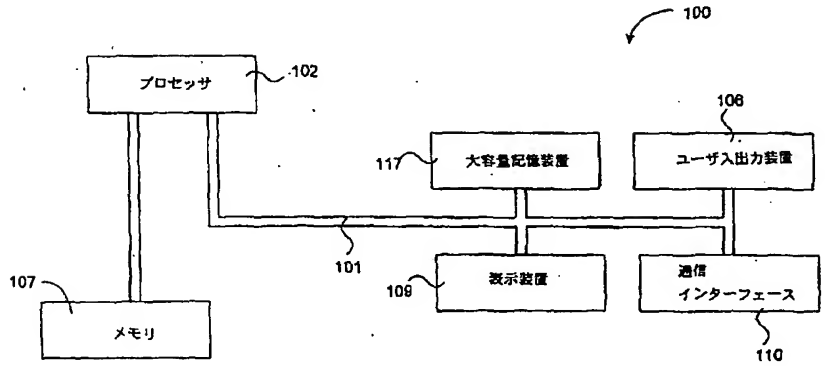
【図１】本発明による手順及び装置を実現するコンピュータ・システムを示している。

【図２】第１の例であるスレッドのスレッド実行タイムラインを示している。

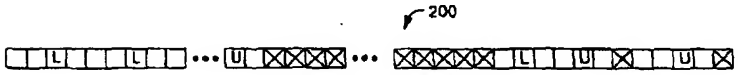
【図３】第２の例である状況のスレッド実行タイムラインを示している。

【図4】第3の例である状況のスレッド実行タイムラインを示している。
【図5】第4の例である状況のスレッド実行タイムラインを示している。
【図6】本発明による同期方法を実現する例示的な構造を示している。
【図7】本発明による同期方法の実現において便利なデータ構造を示している。

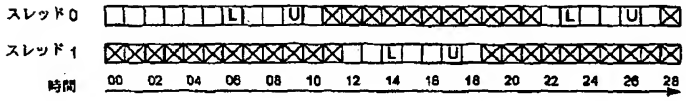
【図1】



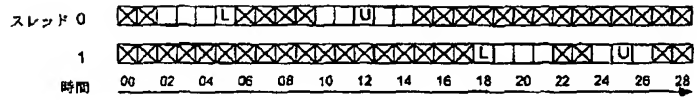
【図2】



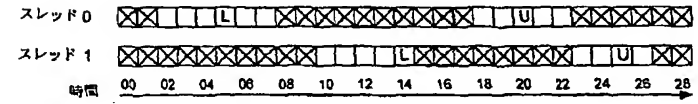
【図3】



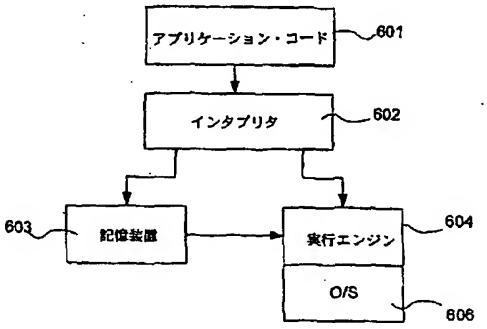
【図4】



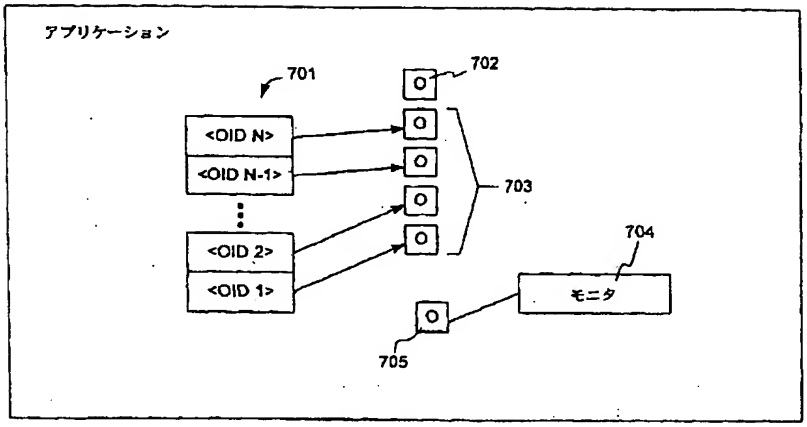
【図5】



【図6】



【図7】



フロントページの続き

(71)出願人 597004720
2550 Garcia Avenue, MS PAL1-521, Mountain View, California 94043-1100, United States of America

【外国語明細書】

1. Title of the Invention: FAST SYNCHRONIZATION FOR PROGRAMS WRITTEN IN THE JAVA PROGRAMMING LANGUAGE

2. Claims

1. A method for execution in a processor having a plurality of threads executing thereon, the threads including synchronized operations that refer to at least one shared object, wherein the shared object is identified by an object identification (OID), the method comprising the steps of:

selecting a first thread of the plurality of threads including a synchronized operation for execution;

upon entering the selected thread, indicating that the at least one shared object should be locked by pushing the OID of the at least one shared object onto a lock stack;

executing the synchronized operations defined by the selected thread; and

removing the indication by pushing the OID from the lock stack.

2. The method of claim 1 wherein during the step of executing the operations defined by the selected thread the selected thread blocks, and the method further comprises a step of locking the at least one shared object after the selected thread blocks.

3. The method of claim 2 further comprising the steps of creating an instance of a monitor object corresponding to the shared object.

4. The method of claim 3 wherein the monitor object comprises a queue header, a counter, and an owner field identifying the selected thread.

5. The method of claim 3 wherein the step of locking comprises creating the monitor object in application memory space.

6. The method of claim 4 wherein the step of locking comprises creating the monitor object in kernel memory space.

7. The method of claim 2 wherein the step of locking further comprises:

determining when the selected thread blocks; and

performing the locking operation in response to the determining step.

8. The method of claim 2 wherein the step of locking further comprises:

determining when the selected thread resumes after the block; and

performing the locking operation only if one other of the threads including a synchronized operation has blocked since the selected thread blocked.

9. The method of claim 1 further comprising the steps of:

selecting a second thread for execution, wherein the second thread includes an operation synchronized with the synchronized operation of the first thread;

upon entering the second thread, determining whether the shared object is locked.

10. The method of claim 9 wherein when it is determined that the shared object is locked the method further comprises the steps of creating a monitor object

associated with the shared object to determine when the shared object is unlocked; and

executing the operations defined by the second thread after it is determined that the shared object is unlocked.

11. The method of claim 9 wherein when it is determined that the shared object is unlocked the method further comprises the steps of indicating that the shared object should be locked by pushing its OID onto the lock stack; and

executing the operations defined by the second thread; and

removing the indication by pushing the OID from the lock stack.

12. A computer system for executing an application comprising a plurality of synchronized threads of execution, wherein each synchronized thread refers to at least one object identified by an object identification (OID) that is shared among a plurality of synchronized threads, the computer system comprising:

a processor;

a memory coupled to the processor;

a multithreading operating system that supports multiple threads of execution in a shared address space of the memory;

a lock stack in the memory, the lock stack comprising a plurality of entries sized to hold an OID;

an instruction interpreter executing in the processor and coupled to receive a selected one synchronized thread and cause the selected thread to execute on the processor, wherein upon entering the selected thread the instruction

interpreter indicates that the shared object should be locked by pushing its OID onto a lock stack and upon exiting the selected thread the instruction interpreter removes the indication by pushing the OID from the lock stack.

13. The computer system of claim 12 further comprising:

a thread block indicator operating within the processor to signal when the selected thread blocks during execution; and

object locking devices operating within the processor and responsive to the thread block indicator to lock the shared object identified by the OID in the lock stack.

14 The computer system of claim 12 wherein the object locking devices comprise an instance of a monitor object corresponding to the shared object wherein the monitor object includes a queue header, a counter, and an owner field identifying the selected thread.

15. The computer system of claim 14 wherein the monitor object is instantiated in an application memory space of the memory.

16. The computer system of claim 14 wherein the monitor object is instantiated in a kernel memory space of the memory.

17. The computer system of claim 14 further comprising:

a thread release indicator operating within the processor to signal when the selected thread releases from a block condition during execution; and

object locking devices operating within the processor and responsive to the thread block indicator to lock the shared object identified by the OID in the lock stack only if another of the synchronized threads has blocked since the selected thread blocked.

18. A computer program product comprising:

a computer usable medium having computer readable code embodied therein for synchronized thread execution in a multithreaded processor, wherein each synchronized thread refers to at least one object identified by an object identification (OID) that is shared among a plurality of synchronized threads, the computer program product comprising:

computer program devices operating in the computer system and configured to cause a computer to select one of the synchronized threads for execution;

computer program devices operating in the computer system and configured to cause a computer to indicate that the shared object should be locked by pushing its OID onto a lock stack upon entering the selected thread;

computer program devices operating in the computer system and configured to cause a computer to execute the operations defined by the selected thread; and

computer program devices operating in the computer system and configured to cause a computer to remove the indication by pushing the OID from the lock stack.

19. The computer program product of claim 18 further comprising:

computer program devices configured to cause a computer to lock the shared object after the selected thread blocks.

20. The computer program product of claim 18 further comprising:

computer program devices configured to cause a computer to create an instance of a monitor object corresponding to the shared object wherein the monitor object comprises a queue header, a counter, and an owner field identifying the selected thread.

21. The computer program product of claim 19 further comprising:

computer program devices configured to cause a computer to determine when the selected thread blocks; and

computer program devices configured to cause a computer to perform the locking operation in response to the determining step.

22. The computer program product of claim 19 further comprising:

computer program devices configured to cause a computer to determine when the selected thread resumes after the block; and

computer program devices configured to cause a computer to perform the locking operation only if another of the synchronized threads has blocked since the selected thread blocked.

23. The computer program product of claim 18 further comprising:

computer program devices configured to cause a computer to create a monitor object associated with a shared object, the shared object being identified by an object identifier (OID), the monitor object comprising state information and

methods to determine when the shared object is unlocked if the shared object is locked;;

computer program devices configured to cause a computer to determine whether the shared object is locked upon entering the second thread;

computer program devices configured to cause a computer to create a monitor object associated with the shared object to determine when the shared object is unlocked if the shared object is locked;

computer program devices configured to cause a computer to indicate that the shared object should be locked by pushing its OID onto the lock stack if the shared object is unlocked;

computer program devices configured to cause a computer to execute the operations defined by the second thread; and

computer program devices configured to cause a computer to remove the indication by pushing the OID from the lock stack.

24. A computer data signal enbodied in a carrier wave comprising:

a first code portion comprising code configured to cause a computer to create a monitor object associated with a shared object, the shared object being identified by an object identifier (OID), the monitor object comprising state information and methods to determine when the shared object is unlocked if the shared object is locked;

a second code portion comprising code configured to cause a computer to indicate that the shared object should be locked by pushing its OID onto the lock stack if the shared object is unlocked;

a third code portion comprising code configured to cause a computer to execute the operations defined by the second thread; and

a fourth code portion comprising code configured to cause a computer to remove the indication by pushing the OID from the lock stack.

3. Detailed Description of the Invention

BACKGROUND OF THE INVENTION

1. Field of the Invention.

The present invention relates, in general, to data processing, and, more particularly, to thread synchronization in JAVA language programs.

2. Relevant Background.

The JAVA™ (a trademark of Sun Microsystems, Inc.) programming language, is an object-oriented programming language developed by Sun Microsystems, Inc., the Assignee of the present invention. The JAVA programming language has found success as a language and programming environment for networked applications. The JAVA programming language is attractive due to its many features, including standardized support for concurrent execution of program threads. The JAVA programming language's concurrency features are provided at both a language (syntactic) level and through a "threads" library. At the language level, an object's methods can be declared "synchronized". Methods within a class that are declared synchronized do not run concurrently and run under control of "monitors" to ensure that variables remain in a consistent state.

Bach time a synchronized method is entered or exited, the JAVA language requires calls to the operating system

(O/S) kernel to allocate thread synchronization resources. Calls to the kernel may require tens if not hundreds of instructions depending on the O/S in use. In comparison, the synchronized method itself may require only a few lines of code. As an example, a dictionary hash table method can be implemented with fewer than ten instructions, but to implement it as a synchronized method requires more than 100 instruction in a typical operating system. Hence, thread synchronization significantly adds to the execution time of many programs.

This overhead is required in programs that make heavy use of multi-threading and depend on thread synchronization. However, this overhead is undesirable in programs that are single-threaded. Similarly, even in multithreaded programs, a large number of the threads may in fact execute correctly without the synchronization overhead. Hence, a need exists for a thread synchronization mechanism that only incurs the overhead associated with O/S thread management resource allocation only when those resources are needed.

Operating systems conventionally enable multithreading in one of two ways: preemptable and non-preemptable. A preemptable thread operating system (e.g., Solaris and Windows/NT) include O/S techniques and devices that enable one thread to interrupt another concurrently executing thread. Hence, at any given time, an executing thread cannot predict whether it will continue to execute or whether it will be blocked by another thread. Hence, the application cannot manage thread synchronization on its own because it lacks visibility as to when threads will be blocked. Preemptable threads are also valuable in

multiprocessing machines to efficiently distribute execution of threads across multiple processors.

Non-preemptable multithreading is an simpler form of multithreading that supports a mode of thread execution, whereby once a thread begins to execute, it cannot be blocked by another thread. A thread may halt or block itself, yield control to other threads, or be blocked by virtue of waiting for input/output (I/O) to complete. There remain a large number of applications that can be implemented as single threads and which do not require the preemptive multithreading features of an operating system. Non-preemptive operating systems will likely exist in information appliances and simpler operating systems for some time. Where the O/S ensures that each thread cannot be preempted, allocation of thread synchronization resources in the O/S increases program execution time with little benefit.

Multithreading takes advantage of parallelism inherent in (or designed into) many programs. However, legacy programs often exhibit little parallelism. Moreover, some programs by the nature of their behavior do not exhibit a high degree of parallelism. These programs are slowed by having to incur the overhead associated with multithreading operating systems without reaping any benefits because of their non-parallel structure. Hence, a need exists for a thread synchronization mechanism that speeds up execution in programs that are essentially un-threaded yet running on a multithreading operating system.

SUMMARY OF THE INVENTION

Briefly stated, the present invention involves a method for synchronized thread execution in a multithreaded processor. Each synchronized thread refers to at least one object identified by an object identification (OID) that is shared among a plurality of synchronized threads. One of the synchronized threads is selected for execution. Upon entering the selected thread, an entry sequence indicates that the shared object should be locked by pushing its OID onto a lock stack. The operations defined by the selected thread are executed and the indication is removed by popping the OID from the lock stack.

The foregoing and other features, utilities and advantages of the invention will be apparent from the following more particular description of preferred embodiments of the invention as illustrated in the accompany drawings. Still other embodiments of the present invention will become readily apparent to those skilled in the art from the following detailed description, wherein is shown and described only the embodiments of the invention by way of illustration of the best modes contemplated for carrying out the invention. As will be appreciated, the invention is capable of other and different embodiments and several of its details are capable of modification in various obvious respects, all without departing from the spirit and scope of the present invention. Accordingly, the drawings and detailed description are to be regarded as illustrative in nature and not as restrictive.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

In general, the present invention involves a method for handling synchronized threads in programs written in the JAVA programming language that postpones initiating the high-overhead thread synchronization structures until it is determined that they are needed to ensure program correctness. In other words, in many applications, particularly single-threaded programs, the application

itself ensures that the programs are deterministic, and the high overhead associated with multi-threaded synchronization serves only to slow program execution. Accordingly, in accordance with the present invention implementation of complex synchronization mechanisms including full data structures is postponed such that they may in fact never be implemented.

Although the problems of thread synchronization are not unique to the JAVA programming language, some features of the JAVA programming language affect the problem. JAVA programming language synchronization involved an O/S assigned lock and monitor object that were created each time a program entered a method labeled "synchronized". The monitor object required data structures that were quite large, including two queue headers, a counter, and an owner field. The size of these structures prohibits them being included in a standard object header. The present invention addresses this difficulty by avoiding the creation of these lock and monitor objects until they are necessary.

The present invention is described in terms of a JAVA programming environment implementation such as a JAVA virtual machine (JVM), just-in-time (JIT) compiler, or a compile time or run time mechanism that converts JAVA programming language code into another programming language. However, the present invention is useful in any programming language that enables statements or program objects to access multithreading features of an operating system. The present invention does not require changes to the programming language itself, but rather provides method for implementing a program such that the program is not forced

to perform unnecessary O/S operations to support thread synchronization.

FIG. 1 illustrates in block diagram form a computer system incorporating an apparatus and system in accordance with the present invention. Processor architectures and computing systems are usefully represented as a collection of interacting functional units as shown in FIG. 1. These functional units perform the functions of fetching instructions and data from memory, processing fetched instructions, managing memory transactions, interfacing with external I/O and displaying information.

FIG. 1 shows a typical general purpose computer system 100 incorporating a processor 102 and using both an application program and an operating system executing in processor 102. Computer system 100 in accordance with the present invention comprises an system bus 101 for communicating information, processor 102 coupled with bus 101 through input/output (I/O) devices within processor 102. Processor 102 is coupled to memory system 107 using a memory bus 103 to store information and instructions for processor 102. Memory system 107 comprises, for example, one or more levels of cache memory and main memory in memory unit 107. It should be understood that some cache memory is included on-chip with processor 102 in most applications in addition to cache and memory in memory system 107.

User I/O devices 106 are coupled to bus 101 and are operative to communicate information in appropriately structured form to and from the other parts of computer 100. User I/O devices may include a keyboard, mouse, magnetic or tape reader, optical disk, or other available I/O devices,

including another computer. Mass storage device 117 is coupled to bus 101 and may be implemented using one or more magnetic hard disks, magnetic tapes, CD ROMs, large banks of random access memory, or the like. A wide variety of random access and read-only memory technologies are available and are equivalent for purposes of the present invention. Mass storage 117 includes computer programs and data stored therein. Some or all of mass storage 117 may be configured to be incorporated as part of memory system 107.

The present invention is particularly useful in operating systems where O/S threads are not preemptable. However, operating systems with preemptable threads can be made to work with the present invention by forcing only one thread at a time to execute. Moreover, the present invention can be used on operating systems that handle both preemptive and non-preemptive threads on a task-by-task basis.

A primary method for thread synchronization in many software languages (including JAVA programming language) is a monitor and lock paradigm. Although the present invention simplifies this locking process, the functionality achieved by the present invention must meet the language specifications so that the threads behave in a predictable manner. A lock operation behaves by first attempting to obtain a lock on a specified object (e.g., the object x in the examples herein). The lock operation will prevent further thread execution until the lock operation is completed (i.e., a lock is obtained). Only after the lock operation completes are the statements in the body of the synchronized method executed. When execution of all the

statements in the body of a given thread are completed or the thread is otherwise terminated, an unlock operation is performed on the same lock.

It should be noted that the O/S overhead associated with the locking mechanism of the prior art can, of course, be avoided simply by not locking any objects, or locking as few objects as possible. However where any two methods access a single object or objects, it is essential to use the locking mechanism. The present invention is directed to a method for efficiently implementing the locking mechanism specified in the application to provide expected application behavior.

Where only one thread of a group of synchronized threads can be executing at a time, as shown in the example of FIG. 2, there is not, in general, a need to create associated monitor data structures when an object is locked. Likewise, where multiple threads can be running, but none can preempt the execution of another, monitor data structures are not generally needed. In these circumstances, the monitor data structures will only be needed when an executing thread is blocked for some time in a manner that is not known to the application itself.

FIG. 2 diagramatically shows a thread 200 generated in accordance with the present invention. Each software application comprises a collection of tasks directed to implement particular application behavior. Each task is in turn implemented by a plurality of threads such as thread 200 shown in FIG. 2. Each thread issues a stream of use, assign, lock, and unlock operations as dictated by the

semantics of the program it is executing. Each atomic operation is represented in FIG. 2 by a box in thread 200.

Of particular interest in the understanding of the present invention a locking operations indicated by a "L", unlocking operations indicated by a "U" and blocking operations indicated by a cross through the operation box. For ease of reference a blocking operation is indicted by a series of operation boxes to indicate duration of the blocking, however, it should be understood that in practice is it is a single operation that blocks for the entire duration. Other operations are generically indicated by blank operation boxes, but these may include a wide variety of operations. Thread 200 may include any number of atomic operations from fewer than ten operations up to several hundred operations, or thousands of operations in an essentially single threaded task. The ellipses in FIG. 2 indicate that any number of non-blocking and blocking operations may be included in a typical thread 200.

In a multithreaded application or task, a plurality of threads 200 are implemented to perform task behavior. In a accordance with the present invention, a multithreaded application is caused to execute in a "serial" fashion meaning that the threads are not preemptable. As described above, this may be imposed by the operating system.

In cases where the operating system allows preemptable threads, the present invention operates to ensure that synchronized thread 0 and thread 1 shown in FIG. 3 - FIG. 5 cannot execute concurrently. This is done, for example, by executing each synchronized method in a construct called a "critical section" such that program control enters a

critical section before executing the synchronized method and no other thread can enter the critical section until the first thread to enter exits.

FIG. 3 - FIG. 5 illustrates various situations during execution of synchronized threads labeled thread 0 and thread 1 and use the same nomenclature and graphical representations described above in reference to FIG. 2. FIG. 3 shows a situation in which thread 0, executes a number of operations, at least one of which accesses a shared object (not shown). Thread 0 is synchronized with thread 1 in FIG. 3 because thread 1, for example, accesses the shared object also. Thread 0 and thread 1 may be related as producer-consumer (i.e., one thread changes the shared object and the other thread uses the changed object) or the shared object may support only one access at a time and so thread 0 and thread 1 are synchronized to avoid conflict.

The lock and unlock operations in each thread are performed at any time during the thread execution upon entering and exiting a critical section of the thread (i.e., a method within a thread containing operations that access the shared object). A thread may execute more than one lock operation before executing an unlock operation (as shown in FIG. 2), but each lock must be followed by an unlock before the shared object is available to another thread. The programming language may allow explicit object locking/unlocking. Alternatively, in the case of the JAVA programming language a "synchronized" statement in the method implies the locking/unlocking operations.

The lock operation is implemented in accordance with the present invention by pushing an object identification (OID) onto a special purpose stack (701 in FIG. 7). In the particular examples herein, the OID is an address of the shared object "x" in memory, although any OID that uniquely identifies a locked object or object is equivalent to the specific implementation. In contrast with the prior art, the lock operation does not cause the OS to allocate resources or create data structures to provide thread synchronization. A thread may acquire multiple locks in this manner, although only a single lock is shown for simplicity. Moreover, the thread may acquire multiple locks on the same object such that each acquired lock must be unlocked before the object becomes available for another thread.

Only when there is an actual monitor object does there need to be any appreciable code executed. This may block the executing thread, if the lock is already granted. A similar procedure may be executed at unlock time resuming a thread. Remembering that the existence of a monitor object is the exception rather than the rule, the entry sequence into a synchronized method is logically:

```
if(monitorExists(o)  ) {
     monitorLockProcessing(o)
} else {
     pushOnStack(o) ;
}
```

and an exemplary exit sequence is:

```
if(monitorExists(o)  ) {
     monitorUnlockProcessing(o)
} else  {
     popStack();
}
```

where monitorLockProcessing and monitorUnlockProcessing represent methods that call conventional lock/unlock operations.

It is important to note that the synchronization technique of present invention does not impose ordering constraints on the execution of thread 0 and thread 1. In other words, thread 1 may execute before thread 0 in any of the examples, and other mechanisms must be employed to constrain order. Examples of these mechanisms in the JAVA programming language include the wait(), notify() and notifyAll() methods available in the JAVA programming language Threads class, although equivalents are available in other program languages.

In the example of FIG. 3, thread 1 and thread 0 are not CPU bound meaning that there are many available CPU cycles as compared to the number of operations that must be completed. In other words, neither thread 0 nor thread conflict with each other during their execution even though they access a shared variable and so are declared synchronized. The number of clock cycles required to execute most instructions is fairly predictable with the notable exception of memory operations, operations that

require input/output (I/O), and instructions that wait for a specified event to occur, and the like. As shown in FIG. 3, thread synchronization is not really an issue because thread 1 and thread 0 do not execute concurrently. In this case, no O/S thread synchronization resources are needed (although they are nevertheless allocated in the prior art). A significant feature of the present invention is that the situation shown in FIG. 3 will not result in expenditure of O/S resources (i.e., creation of an O/S monitor object) even though the application specified synchronous threads.

FIG. 4 shows a related situation in which, even though thread 0 and thread 1 block while an object is locked during execution, they do not in fact conflict. The blocking in thread 0 occurs between operations 6 and 9 and may be caused by, for example, an I/O operation. At operation 5 a lock is placed on the shared object, hence, during the blocking phase the shared object remains locked. However, because thread 1 does not attempt to access the shared object until its own lock operation at operation 18, there is no conflict. As in the prior example, conventional processing would have expended OS resources to create monitor objects at time 5 even though none are in fact needed.

In accordance with a first embodiment of the present invention, the shared object's OID is pushed onto stack 701 (shown in FIG. 7) when thread 0 first blocks after the lock is in place (i.e., time 6), and the OID is removed when the thread resumes at time 10. This first embodiment results in minimal resource expenditure as the lock is implemented without a call to the OS by simply pushing an OID onto stack 701. In accordance with a second embodiment, the system

delays pushing the OID onto stack 701 until 1) thread 0 resumes, and 2) the resumed thread is not the thread that last blocked. Hence, in the second embodiment a lock is never implemented in the example of FIG. 4 because thread 0 resumes before any other thread has created a lock.

In contrast, FIG. 5 shows an example including thread 0 that blocks at operation 8 during its execution. In FIG. 5, the blocking operation in thread 0 resumes at operation 18. At operation 10 thread 1 unblocks and resumes execution until operation 14 in which it attempts to obtain the lock on the shared object. In this case, the shared object is already locked by thread 0 and conventional lock/monitor processing is required to notify thread 1 when it can continue. In accordance with the present invention, thread 1 is blocked by the monitor object created during the time indicated by the bold-lined boxes 15-22. At time 23, thread 0 blocks again, but this time the shared object is unlocked, enabling thread 1 to resume.

In the first embodiment, allocation of thread synchronization data structures is delayed until operation at time 8 when thread 0 blocks. In accordance with the second embodiment, the OID is not placed on stack 701 until thread 1 unblocks at time 10. Hence, only when the lock/monitor resource is actually needed are the resources allocated. In the first embodiment, aside from the delayed allocation of resources, thread 0 and thread 1 execute in a manner substantially identical to prior synchronization methods. The first embodiment provides improved performance because synchronization resources are not allocated at all in situations resembling that shown in FIG. 3 and FIG 4.

In the second embodiment, allocation of lock/monitor resources is postponed until a thread resumes. In this case, thread 1 is blocked by the application itself because it is now aware of the lock on the object because stack 701 is maintained in application or user memory space. When thread 1 resumes at time 10, it determines if it was or was not the last thread that blocked without a call to the O/S. In FIG. 5, thread 1 was not the last thread that blocked, hence, monitor objects are required to synchronize thread 1 with thread 0. In the example of FIG. 4, however, when thread 0 resumes it was the last thread that blocked, hence, monitor resources are not required. The longer it is possible to delay allocation of the synchronization resources the more likely it becomes that one of the conflicting threads will unlock the shared object and synchronization will be unnecessary.

Another advantage of the present invention is that monitor resources can be implemented by without involving operating system resources. As described in reference to FIG. 5, the unblocking of thread 1 at operation 10 can be handled without the OS because the information that the shared object is locked is held in application memory space. While the present invention can be implemented using conventional OS synchronization resources, it is desirable to implement the monitor object in application memory space to avoid the tens or hundreds of operations that may be required to implement the monitor objects in kernel memory space using OS synchronization resources.

FIG. 6 shows fundamental software or hardware implemented devices that implement the present invention.

Computer system 100 (shown in FIG. 1) executes application code 601 using an interpreter 602. Application code 601 includes single threaded or multithreaded code in a conventional language such as the JAVA programming language. Interpreter 602 receives the application code and generates executable instructions that can be executed by execution engine 604. The executable instructions may include calls to OS 606, or may manipulate data in memory 107 directly. Interpreter 602 may be implemented in the form of a compiler that is operable prior to run time (i.e., at compile time) or operable at run time such as a just-in-time compiler provided with the JAVA virtual machine. The executable instructions from interpreter 602 may also be stored in storage 603 (e.g., magnetic media, optical media and the like) for later execution by execution engine 604 at run time.

Referring to FIG. 7, a plurality of objects 702, 703, and 705 are shown. Objects 703 are locked by a corresponding entry in stack 701. In contrast, object 705 is locked by a monitor object 704 created in a conventional fashion. The fact that an object is locked and the number of times it is locked is recorded by execution engine 604 (shown in FIG. 6) by pushing the address of the object (indicated as object ID or OID in FIG. 7) onto stack 701 at lock time. The entry is popped from stack 701 at unlock time and the object associated with the popped OID becomes an unlocked object 702. In this way, if the thread does not block while an object is locked, there is nothing more than needs to be done to ensure thread synchronization. If, however, the thread does block then the objects on stack 701

can be allocated monitor data structures 704 that are then used in a traditional way and stack 701 emptied.

As noted above, on the rare occasions that real lock/monitor data structures are created they can be implemented without the involvement of operating system 606. Memory 107 is allocated into application or user memory space 701 and kernel or OS memory space 702.

Although the invention has been described and illustrated with a certain degree of particularity, it is understood that the present disclosure has been made only by way of example, and that numerous changes in the combination and arrangement of parts can be resorted to by those skilled in the art without departing from the spirit and scope of the invention, as hereinafter claimed.

4. Brief Description of the Drawings

FIG. 1 shows computer system implementing the procedure and apparatus in accordance with the present invention;

FIG. 2 shows a thread execution timeline of a first example thread;

FIG. 3 shows a thread execution timeline of a second example situation;

FIG. 4 shows a thread execution timeline of a third example situation;

FIG. 5 shows a thread execution timeline of a fourth example situation;

FIG. 6 shows an exemplary structure for implementing the synchronization method in accordance with the present invention; and

FIG. 7 shows data structures useful in implementation of the synchronization method in accordance with the present invention.

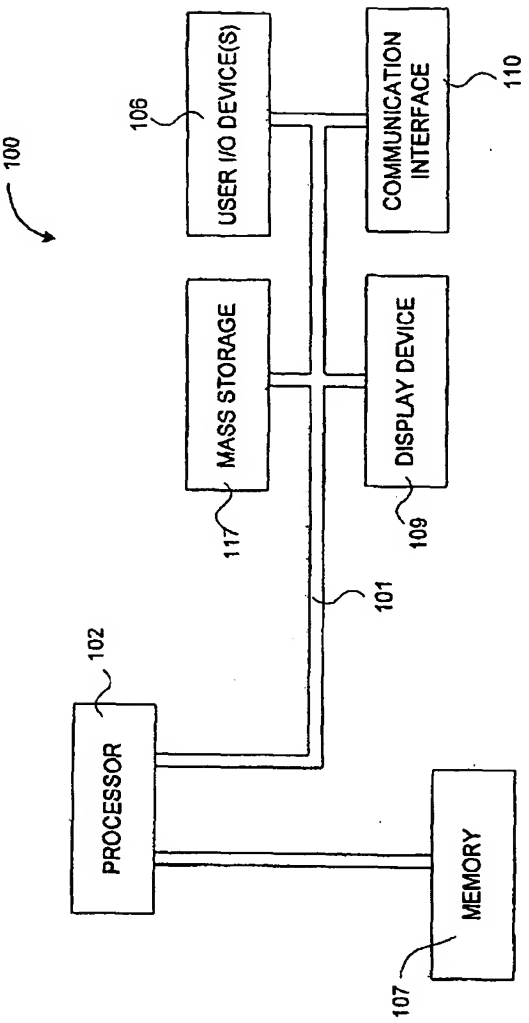

100
USER I/O DEVICE(S)
106
COMMUNICATION INTERFACE
110
MASS STORAGE
117
DISPLAY DEVICE
109
101
PROCESSOR
102
MEMORY
107

FIG. 1

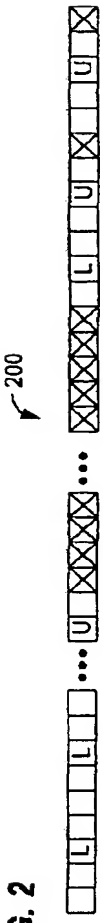


FIG. 2

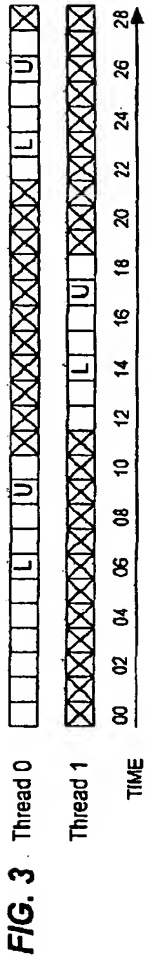


FIG. 3

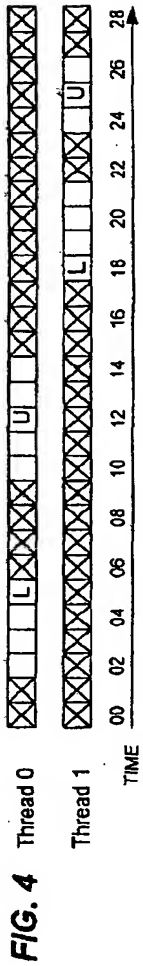


FIG. 4

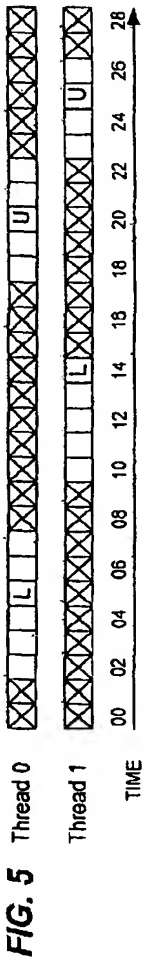
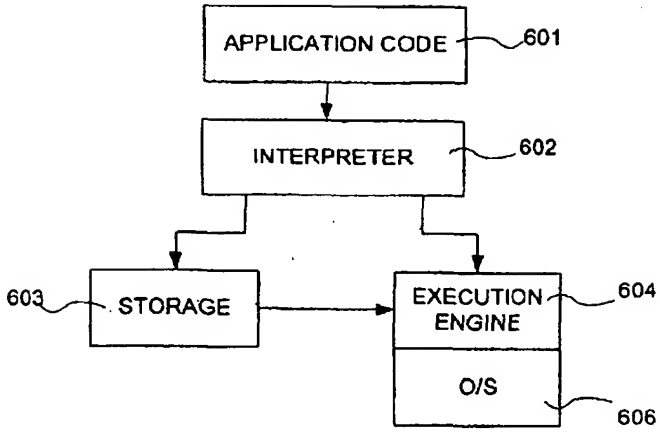


FIG. 5

FIG. 6



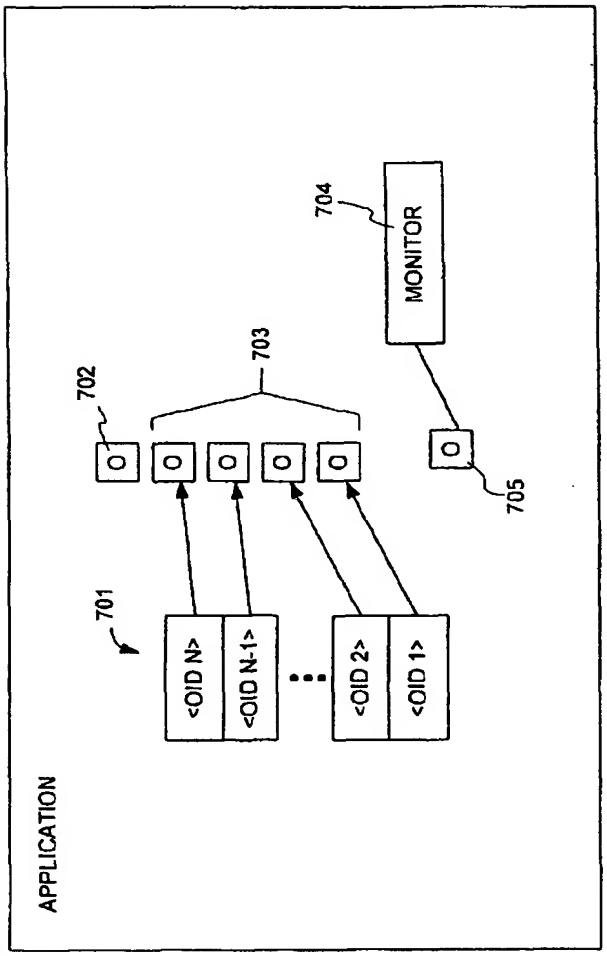


FIG. 7

1. Abstract

A method, system, and computer program product for synchronized thread execution in a multithreaded processor are described. Each synchronized thread refers to at least one object identified by an object identification (OID) that is shared among a plurality of synchronized threads. One of the synchronized threads is selected for execution. Upon entering the selected thread, an entry sequence indicates that the shared object should be locked by pushing its OID onto a lock stack. The operations defined by the selected thread are executed and the indication is removed by pushing the OID from the lock stack.

2. Representative Drawing: Fig. 1